ECSE426 - Microprocessor Systems
Group 20 - Lab 3 & 4

Fabrice Normandin 260636800
Marcel Morin 260605670

March 20, 2018

Abstract

This report outlines the engineering process we used while completing Labs 3 & 4. These most recent labs built upon the ADC sampling, filtering and 7-segment display work done previously as part of Lab 2. The main objective of this lab was to create a system which accepts user input through a keypad, and then attempts to match the given value (a RMS voltage) by using a PWM signal. This system also had to successfully manage the transitions between the different states of operation (sleep, keypad input, voltage matching), all-the-while conserving CPU resources as much as possible. The system was first created using an interrupt-driven approach (Lab3), and later using the multithreaded paradigms of the CMSIS_RTOS framework (Lab4). Despite some portions of this lab presenting a significant technical challenge, it can be considered a resounding success, as all of its objectives were successfully completed. While our current solution makes a great effort towards CPU efficiency, additional measures could be taken in order to reduce the power consumption even further.

Contents

1 Problem Statement

A comprehensive list of all system requirements can be found within the Lab 3 & Lab 4 handouts. Among these requirements, one of the most challenging to meet was one related to the keypad, as it showed that the system had to respond differently to buttons depending on their press duration, and generate a transition between different modes of operation (a long "*" press would put the system into sleep mode, for example). Consequently, an additional requirement was created, by which the system code had to be structured as a finite-state machine (FSM), and successfully manage state transitions in a clear, well-structured way. State diagrams were also deemed essential, and we considered it a system requirement that each sub-system have its own related state diagram, in order to clearly represent the functioning of our system.

2 Theory and Hypothesis

2.1 PWM Signals

One of the major theoretical components of this experiment is the Pulse Width Modulation (PWM) signal which, when applied across a load, generates a corresponding RMS voltage. The PWM has a set period, which needs to be faster than the time constant of the circuit, in order to allow the capacitor to retain some charge between successive pulses. Otherwise, the capacitor would discharge completely between each and every PWM cycle, which would cause the overall RMS voltage not to reach the desired value reliably.

The time constant of our RC circuit, with our resistor and capacitor having values of $4.7k\Omega$ and $0.1\mu F$ respectively, was calculated to be approximately 0.47ms, using Equation 1.

$$\tau = RC = 0.1\mu F * 4.7k\Omega = 0.47ms \tag{1}$$

The duty cycle of a PWM signal represents the fraction of the period during which the output signal is high. By setting the duty cycle to 0, there would be no active portion in the cycle, which would yield a calculated RMS value of 0 volts. By progressively incrementing the duty cycle, we can expect to start to measure an increasing RMS voltage across the capacitor, as it begins to receive periodic packets of charge during the active duty cycle of our PWM pulse, and then dissipates its charge during the rest of the PWM cycle. By choosing an appropriate value for the duty cycle, we can create an equilibrium where the capacitor gains as much charge during the voltage pulse as it dissipates during the rest of the cycle[1], effectively maintaining an average voltage over time. Figure 1 shows an example of such an equilibrium.

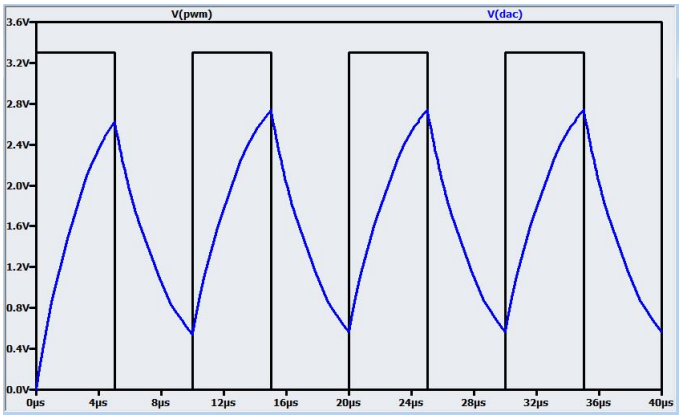


Figure 1: The general shape of a PWM signal in charging/discharging a RC circuit.
Source: https://www.allaboutcircuits.com

2.2 Making a PWM Controller

With our PWM timer created, we need a controller to compare our measured RMS with the target RMS value given by the user via keypad. This controller would then see if the duty cycle needed to be increased, decreased, or kept at the same value, based on if the measured RMS value was above, bellow, or less than `THRESHOLD` away from our target. The implementation of this controller could be quite simple. as it just needs to compare two values in order to determine which action to perform. Here is some pseudocode for such a controller.

```
if(ABS(current_RMS - target_RMS) <= THRESHOLD){
        //Don't change anything. We reached the target voltage.
}else if(current_RMS < target_RMS){
        //increase Duty Cycle
}else{
        //decrease Duty Cycle
}
```

[1] Pulse-width modulation: Wikipedia

2.3 Choosing the PWM and ADC frequencies

With our time constant of 0.470 ms calculated in Equation 1, it is required that the PWM timer period be smaller than this value. Hence, our prediction is that by using a 0.1ms period for our PWM timer, we should be able to successfully generate the required RMS voltage across the capacitor.

Given the corresponding PWM timer frequency of 10KHz, it is important to choose an appropriate value of ADC sampling frequency in order to ensure that we can detect the changes applied by our controller, as well as reliably measure an "average" voltage on the circuit over time, in the form of the RMS of a number of samples. In order to do so, our ADC sampling period needs to be much longer than our PWM signal period, such that the ADC samples are somewhat "randomly" distributed along the capacitor voltage curve (an example of which can be seen in Figure 1 above). With multiple random samples taken across multiple cycles, our converted ADC values will give an accurate combined RMS voltage value. To that end, we concluded that using the same ADC sampling frequency of 1kHz as in Lab 2 would probably yield positive results.

Following equation 2, the `prescaler` and `period` settings of the ADC will be picked to meet the above requirements and generate a 1 kHz sampling frequency.

$$Timer freq. = \frac{Clock freq.}{(prescaler + 1) * period} \tag{2}$$

2.4 Multithreading

Another requirement of this lab is to implement threads into our system. We did this using the FreeRTOS (Free Real Time Operating System) kernel. This makes available a middleware that is able to schedule and handle the creations of threads and free up the CPU. The advantages of running multiple threads concurrently means that our display thread (which takes care of refreshing our display with the appropriate user input value or measured RMS value of our circuit) and our keypad thread can share CPU time, but exist as separate entities, each with their corresponding logic and program space. Another advantage of running threads in our system is that we are no longer dependent on the internal SysTick Interrupt Handler as a provider of time. The ADC can also be modeled as a thread which waits until a `buffer_full` flag indicating that the DMA buffer was filled is set, then compute the RMS value, and go back to sleep. Whether to implement this in a thread or ISR would be logically equivalent, since either these operations are always in the same sequence. The choice of paradigm does make a difference though, as will be explained in the Testing and Observations section.

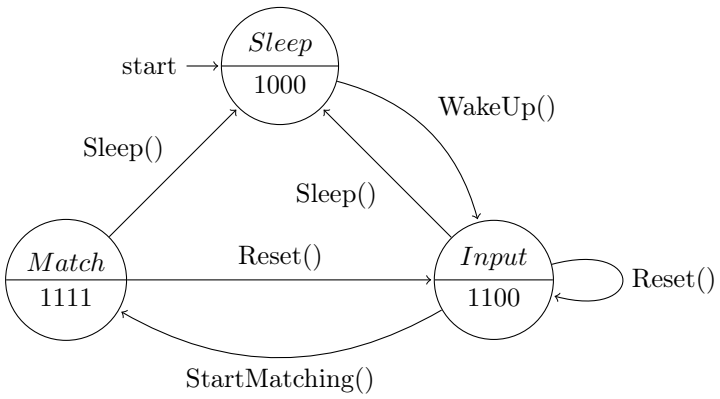
3 Implementation

This section will describe the design and engineering processes that were used while implementing our system, based on the requirements described above. In order to simplify and structure its description, the system will first be broken down into its different logical components.

3.1 High-Level System Architecture

The various requirements and corresponding program functionality of our system can be broken down into four main areas: the 7-segment display, the keypad, the ADC and the PWM controller. The keypad and display components were implemented as Threads, each behaving as a simple finite-state-machine. The ADC and PWM-related logic components were implemented as a part of an Interrupt-Service-Routine (ISR).

Despite the overall system having significant complexity, its high-level behaviour can be broken down into a very limited number of states. The transitions between each state were also well defined (as part of the Problem Statement section). By taking advantage of this fact, a very simple state machine was created, where each state manages to turn each sub-component "on" or "off". A state diagram showing the three system states, along with the name of their transitions and their outputs can be seen in Figure 2.



Output: [Keypad|Display|ADC|PWM]

Figure 2: System high-level state diagram

Sleep	Input	Match
The sleep state is the initial state of the system. In this state, the only active component is the keypad thread. All other components (display, ADC, PWM timer) are turned off, in order to conserve energy. Whenever a keypress is detected, the system transitions into the 'input' state.	During the "input" state, the Keypad thread detects button presses and updates the target value accordingly, while the Display thread shows the current target value. The Keypad and Display threads are active, while the ADC and PWM timer are turned off. Whenever a valid voltage value is entered, the system transitions into the "match" state.	When the system enters the "match" state, the PWM timer and ADC are both started. After each `HAL_ADC_ConvCpltCallback()` interrupt service routine, the ADC samples are filtered and their RMS is extracted. This value is fed to the PWM controller, which modifies the period of the PWM timer in order to attempt to reach the target voltage.

The associated code for this finite state machine can be found within the fsm.c file. Every major component of the system will now be examined individually.

3.2 Display Thread

The logic used in the Display thread was almost all created as part of Lab 2. A state diagram of the display FSM can be seen Figure 3. Its main logic, as shown in Figure 4, can be found within the display_thread.c file.

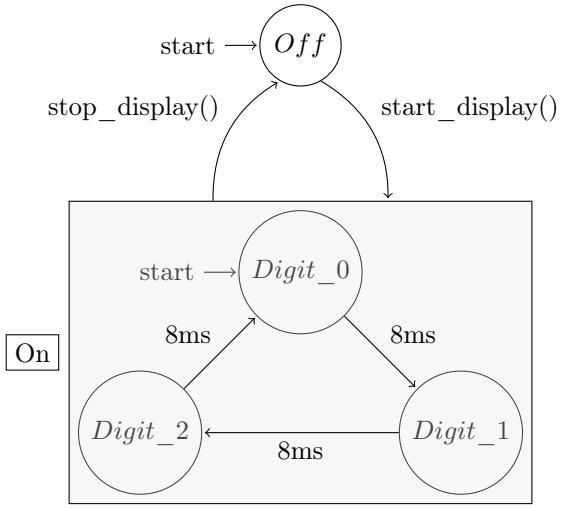


Figure 3: Display thread state diagram

```
void StartDisplayTask(void const * arguments){
  // Which digit is currently active.
  static uint8_t currently_active_digit = 0;

  while(true){
    osSignalWait(display_on, osWaitForever);
     while(display_on){
       // while the display is on, refresh it.
       refresh_display(currently_active_digit);
       osDelay(DISPLAY_REFRESH_INTERVAL_MS);
       currently_active_digit++;
       currently_active_digit %= 3;
     }
     // DISPLAY IS NOW OFF!
     RESET_PIN(DIGITS_0);
     RESET_PIN(DIGITS_1);
     RESET_PIN(DIGITS_2);
  }
}
```

Figure 4: Main Display thread logic

This thread calls the `osSignalWait` function in order to be blocked until the `display_on` variable is set externally, at which point the thread is resumed, and the refreshing cycle begins. Refreshing the display consists of setting the pin associated with the currently active digit high, along with the required segment pins for the corresponding digit of the `displayed_value` variable. This variable may contain either the target value, if the system is currently in the `"Input"` state, or the current RMS, when in the `"match"` state. Since there is no contention for the display (i.e. the system may only be in either state at any time), the use of a synchronization variable - for instance, a semaphore - is not needed. After a digit is set, the thread goes to sleep for `DISPLAY_REFRESH_INTERVAL_MS` (8ms), in order to reduce power consumption. This interval was determined by experimentation, since it is the lowest interval at which the display does not appear to flicker. The effective display refresh rate is therefore $1/(3 * 8ms) \approx 42$Hz.

3.3 Keypad Thread

The keypad was without a doubt the most challenging portion of this lab. The complexity of this component stems from the need for the system to debounce each keypress, as well as keep track of the amount of time each key has been pressed for.

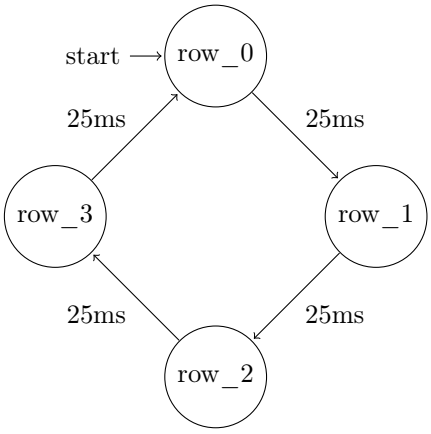


Figure 5: Display thread state diagram

Keypad Output			
Row\Column	col_0	col_1	col_2
row_0	"0"	"1"	"2"
row_1	"3"	"4"	"5"
row_2	"6"	"7"	"8"
row_3	"*"	N/A	"#"

Figure 6: Output of the keypad FSM, depending on which row is currently set, and which column read returns a '1'. Note that the '0' key on our keypad was dysfunctional.

The keypad logic shown in Figure 7 is separated into two distinct steps. First, the `check_for_digit_press()` function is called, which sets and increments the `current_row` variable (represented as `"row_i"` in the FSM diagram of Figure 5), and reports the most recently

observed button press - or the absence of a press - by calling the `keypad_update()` function with the corresponding character, or a whitespace character if no button press was detected in all four rows.

```
void StartKeypadTask(void const * arguments){
 static char pressed_char = NULL;
 while(true){
  pressed_char = check_for_digit_press();
  if(pressed_char != NULL){
   keypad_update(pressed_char);
  }
  osDelay(CHECK_FOR_DIGIT_PRESS_INTERVAL_MS);
 }
}
```

Figure 7: Main Keypad thread logic

Then, the `keypad_update()` function keeps track of the currently pressed digit and counts the number of successive updates received for the current digit. Whenever the number of consecutive updates reaches one of the defined thresholds, the appropriate action is taken, depending on the currently pressed digit. Three thresholds were defined within the keypad_thread.h file: `min_updates_for_change`, `min_updates_for_restart`, and `min_updates_for_sleep`. These three constants represent the minimum number of updates that are required in order to make a state transition. They were calculated using the corresponding desired delays in milliseconds divided by the time required to scan the entire keypad.

Using this mechanism, the Keypad thread can effectively keep track of the currently pressed digit, the time it has been pressed for as well as any press/release, with a time resolution of $4 * 25ms = 100$ms (the time required to scan all rows of the keypad).

Whenever a numeric digit is entered, it enters a shift-register array, `digits`, which holds up to three digits, allowing for a voltage value with 2 decimal points. Whenever the `"#"` key is pressed and the float value formed by the contents of the current `digits` array is valid ($0.5V \leq value \leq 2.5V$), this corresponding float value is assigned to the global `target_voltage` variable, which the PWM controller then tries to match.

The Keypad thread behaves somewhat similarly to the Display thread, as it executes a periodic task before going to sleep for a fixed delay using `osDelay()`. However, the Keypad thread does not use an external signal to turn it on/off, and consequently relies on a "polling" mechanism in order to detect a key press. As will be discussed in the Testing and Observations section, this arrangement is not exactly optimal, as using a combination of an external interrupt to detect the press and a thread to count the length of the press would have been superior in terms of power efficiency.

3.4 ADC Component

Overall, the ADC component of this Lab was not much different from that of Lab 2 (See the appendix). One difference is that the only truly relevant measure was the RMS, and the max and min values were discarded. Another noteworthy difference in this implementation was the use of an external timer (Timer2) to trigger the ADC's conversions. As per in Lab 2, DMA was used, with one `HAL_ADC_ConvCpltCallback()` ISR occurring once 50 values had been gathered.

The ADC frequency we chose to use was 1kHz. This value was chosen because it is significantly slower than the PWM signal frequency, a requirement which was described in the Theory and Hypothesis section. Given that the clock source for timer 2, the ABP2 peripheral clock, is $84MHz$, in order to achieve a 1kHz frequency for Timer2, its prescaler and period settings were set to 83 and 1000, respectively.

The distinction between the ADC and PWM controller components is more for sake of clarity, since they represent different logical modules. However, they are both called sequentially as part of the `HAL_ADC_ConvCpltCallback()` ISR.

One other idea which was explored was to have the ADC be implemented as a thread. The ADC thread would then behave very similarly to the Display thread, as it would simply wait on a `buffer_full` signal before executing the `ADC_buffer_full_callback()` function. However, as

will be discussed in the Testing and Observations section, we ran into significant issues.

Here is a brief breakdown of the different steps involved in the ADC/PWM controller flow.

1. the ADC is started in DMA mode using `HAL_ADC_Start_DMA(...)`.
2. Once 50 values have been acquired, an interrupt is raised.
3. The `HAL_ADC_ConvCpltCallback()` callback is called.
4. The 50 ADC samples are filtered, using the `FIR_C()` function from Lab 1.
5. The RMS value is calculated using the `asm_math` assembly procedure written in Lab 1.
6. The RMS value is fed to the PWM controller, which compares it to the current target value and adjusts the duty cycle of the PWM timer (Timer3) accordingly.

3.5 PWM Controller

The PWM timer was chosen to be Timer3. As previously discussed in theTheory and Hypothesis section, we chose to have a PWM timer frequency of 10 kHz. To maximise our available resolution with respect to the duty cycle, which allows us to match the target voltage more closely, we opted for a `prescaler` value of 0. Using Eq. 2, this gives us a `period` value of 8400, which is the biggest period possible for our desired frequency of 10kHz.

Figure 8 shows a basic representation of the PWM controller logic.

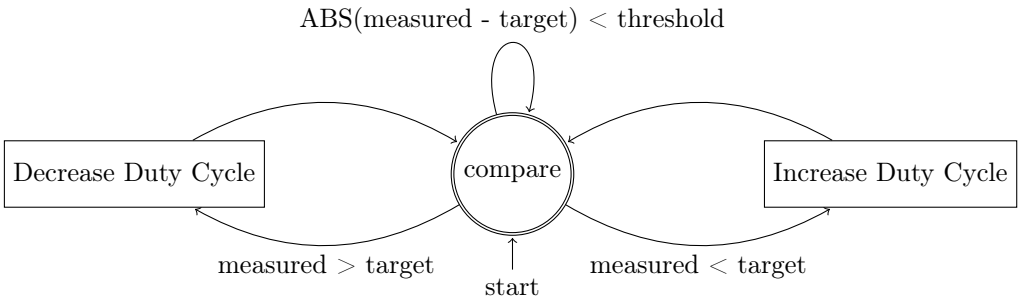


Figure 8: High-level diagram of the PWM controller logic.

During the implementation of this lab, two different PWM controllers were implemented. The first, shown in Figure 9, changes the duty cycle proportionally to the difference between the current RMS voltage and target RMS values. This is the controller that was used during the demo. The second, shown in Figure 10, is significantly more complex. It functions basically as a Successive-Approximation Register[2].

A comprehensive discussion of the performance and tradeoffs associated with each controller can be found within the Testing and Observations section.

[2]Understanding SAR ADCs: Their Architecture and Comparison with Other ADCs

```
/** @brief Controller which adjusts the PWM duty cycle
* in order to match the current target RMS voltage.
* @param current_rms: The current RMS voltage from the ADC.
*/
void adjust_duty_cycle(float current_rms){
  extern float target_voltage;
  // a damping constant, that limits the rate of change of the percentage.
  static const float damping = 0.005f;

  static float current_percentage;
  static int current_period;
  static float difference;

  difference = current_rms - target_voltage;

  current_percentage -= damping * difference;
  current_percentage = BOUND(current_percentage, 0.f, 1.f);


  current_period = round(current_percentage * PWM_TIMER_PERIOD);

  set_pwm_duty_cycle(current_period);
}
```

Figure 9: First implementation of a PWM controller. This controller adjusts the duty cycle by changing it proportionally to the difference between the current and target values.

```
/** @brief Controller which adjusts the PWM duty cycle
* in order to match the current target RMS voltage.
* @param current_rms: The current RMS voltage from the ADC.
*/
void adjust_duty_cycle_2(float current_rms){
  extern float target_voltage;
  static int pwm_period;
  static float last_target_voltage = 0.f;
  const float threshold = 0.01f;
  static int i = 1;

  if(target_voltage != last_target_voltage){
    printf("New target voltage detected: %1.2f\n", target_voltage);
    // restart the 'binary-search' process.
    i = 1;
    last_target_voltage = target_voltage;
  }
  float difference = current_rms - target_voltage;

  if (ABS(difference) < threshold)
  {
    printf("Done. we matched. (difference is %1.5f)\n", difference);
  }
  else if (i >= 32)
  {
    printf("We can't match this voltage! (%1.2f)\n", target_voltage);
    printf("Did the circuit change ?)\n");
    if(ABS(difference) >= 0.20f){
      // if the difference is large, start over.
      printf("Starting over, maybe this will work!\n");
      i = 1;
    }
  }
  else {
    if (difference < 0)
    {
      // we undershoot.
      pwm_period += MAX(PWM_TIMER_PERIOD >> i, 1);
      i++;
    }
    else
    {
      // we overshoot last time. We have to undo the change we did last time
      // (reset that bit).
      pwm_period -= MAX(PWM_TIMER_PERIOD >> (i-1), 1);
      pwm_period += MAX(PWM_TIMER_PERIOD >> i, 1);
    }
    i = BOUND(i, 1, 32);
    pwm_period = BOUND(pwm_period, 0, PWM_TIMER_PERIOD);
    set_pwm_duty_cycle(pwm_period);

  }
}
```

Figure 10: Second implementation of the PWM controller. More complex than the first, this controller uses a "binary-search" approach, similar to a SAR.

4 Testing and Observations

4.1 Keypad Debouncing

Our first component that required testing was the keypad, which unfortunately did not have a formal datasheet available. An important required step for us to figure out which output pin was set by the buttons on the keypad. Our approach to debugging this was to connect what was presumed to be the column pins to the digit pins of the display, and then to press the buttons and observe which segment would light up. Once the keypad pins were mapped and each button was assigned a combination of COL and ROW pins, our focus shifted towards debouncing the digit presses. This was achieved using a deboucing interval of 200ms, in combination with a minimum update count, which is described in Section **??**. During testing, we observed that this debouncing mechanism worked without problems.

4.2 ADC Thread

We initially implemented an additional thread for our ADC that would wake up as soon as the `DMA_buffer_full()` callback was executed by setting a flag. This through testing however showed that the ADC thread would end up blocking the other threads from executing and eventually block our whole program. Figure ? bellow shows our program output and shows the display and keypad threads getting called until only the ADC thread output is shown.

```
ADC Thread
Current voltage: 0.524, Target Voltage: 1.230, current percentage: 2.00066%, current_period: 34 / 1680
Display thread
Display thread
Display thread
Keypad thread
Display thread
Display thread
Display thread
Keypad thread
Display thread
ADC Thread
Current voltage: 0.662, Target Voltage: 1.230, current percentage: 2.28452%, current_period: 38 / 1680
Display thread
Display thread
Keypad thread
Display thread
Display thread
Display thread
Keypad thread
Display thread
ADC Thread
Current voltage: 0.757, Target Voltage: 1.230, current percentage: 2.52114%, current_period: 42 / 1680
Display thread
Display thread
Keypad thread
Display thread
Display thread
Display thread
Display thread
Keypad thread
ADC Thread
Current voltage: 0.821, Target Voltage: 1.230, current percentage: 2556.72551%, current_period: 46 / 1680
Display thread
ADC Thread
Current voltage: 0.890, Target Voltage: 1.230, current percentage: 2.89546%, current_period: 49 / 1680
ADC Thread
Current voltage: 0.933, Target Voltage: 1.230, current percentage: 3.04416%, current_period: 51 / 1680
ADC Thread
Current voltage: 0.974, Target Voltage: 1.230, current percentage: 3.17198%, current_period: 53 / 1680
ADC Thread
Current voltage: 0.993, Target Voltage: 1.230, current percentage: 3.29064%, current_period: 55 / 1680
ADC Thread
Current voltage: 1.016, Target Voltage: 1.230, current percentage: 3.39758%, current_period: 57 / 1680
```

Figure 11: An example of the very weird behaviour observed when using an ADC thread. The ADC, Keypad and Display threads are coexisting peacefully, up until the point at which the ADC thread decides it will keep the CPU all for itself.

To overcome this blocking thread we decided to just keep our ADC DMA processing in a service routine called after every full buffer callback instead of relying on raising the flag to wake up the ADC thread. With this implementation the keypad and display threads would work properly without the blocking that previously occurred.

4.3 PWM Controller Testing

The next component to test was our PWM wave generated by timer 3. In order to make sure our PWM pulse had the correct shape, we connected the oscilloscope to the positive terminal of the diode and plotted the resulting wave form, showed in Figure 12.

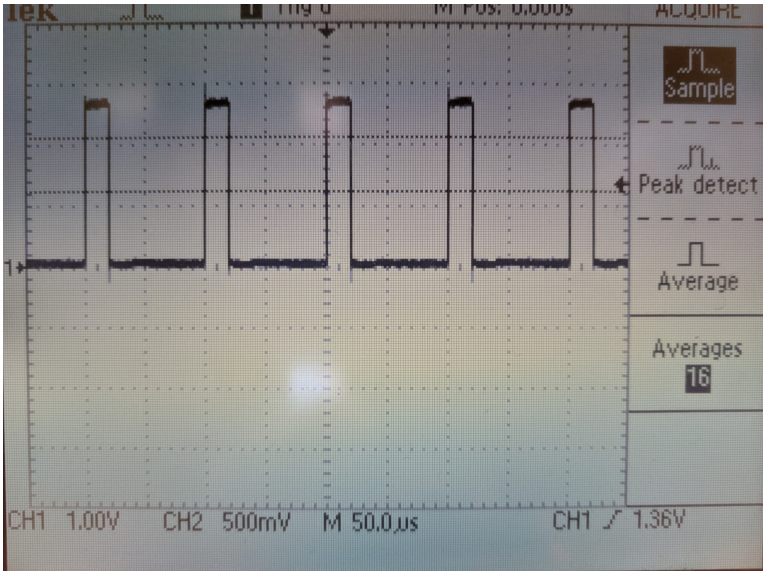


Figure 12: Sample of the PWM signal. The duty cycle appears to be around 25%.

Our first controller, whose code was previously shown in Figure 9, was modified from the idea first introduced in Section 2 (Theory and Hypothesis). It can be seen working in 13. Because this controller adjusts the duty cycle proportionally to the difference between the measured and target RMS values, it fails to reach the target RMS value near the end of the adjustments, partly due to the fact that the difference grows smaller, hence the change in duty cycle also becomes smaller.

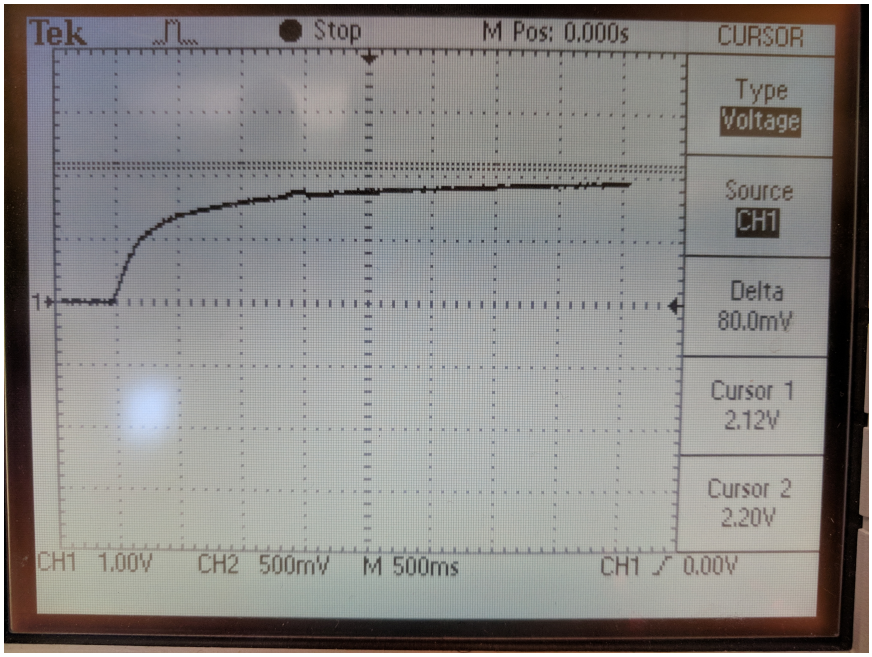


Figure 13: Sample of the first PWM controller in action. As can be observed, the controller never reaches the target value of 2.22V (indicated by cursor 2), despite the sample being $8 * 500ms = 4s$ long.

In contrasts to the first implemented controller, our second one uses a Binary Search approach that tends to overshoot and re-correct as seen in Figure 14. Although this controller is much faster than our first one, it does overshoot, which may not be ideal depending on the direct implementation this controller may be used in. For instance, if this PWM signal served to power a motor, the motor might jerk before reaching the desired speed, which is very undesirable. For the purposes of this project, this behaviour is tolerable, as it does not violate any requirement.

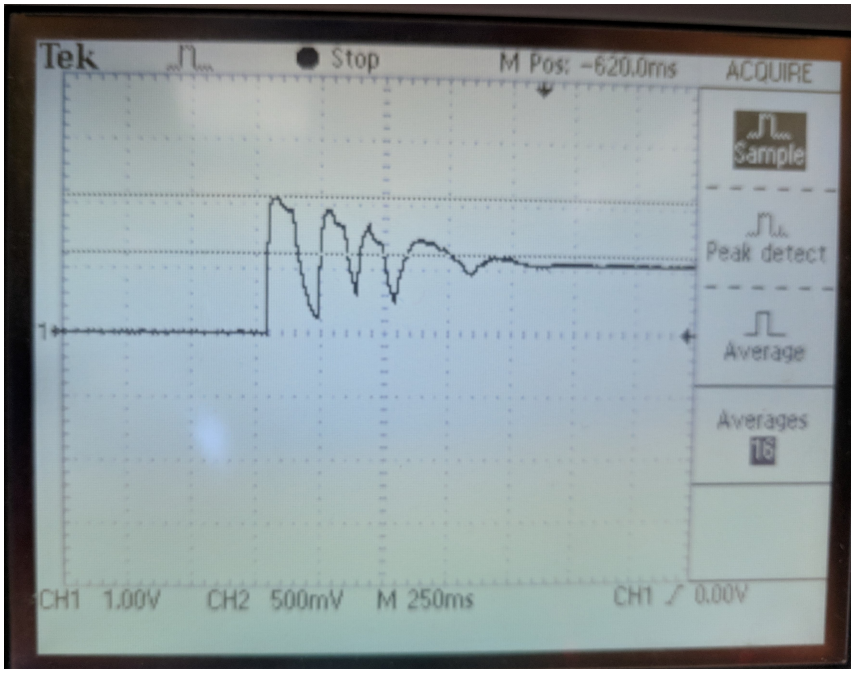


Figure 14: Sample of the second PWM controller in action. This controller manages to settle at the desired voltage in about $5 * 250ms = 1.25s$, a very significant improvement upon the first controller.

4.4 Keypad Thread Improvement

The Keypad thread could be improved in terms of power consumption. In its current state, the keypad thread is using a "polling" mechanism, repeatedly checking each column's GPIO pin to detect if any of them becomes high. Instead, a simple configuration could be devised where all three column pins are each connected to another GPIO pin using diodes. Then, an external interrupt could be used on this pin to wake up the Keypad thread whenever either line becomes high. The keypad thread could then begin polling, counting the time the key has been pressed, up until it is detected that the key has been released. Such a circuit can be seen in Figure 15.

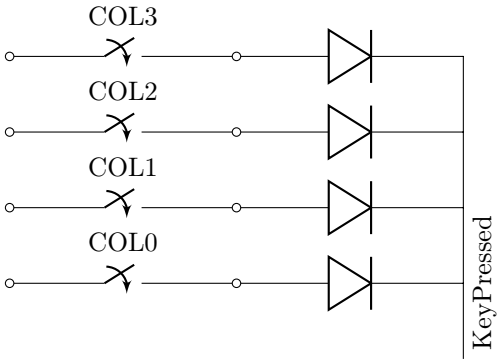


Figure 15: A simple circuit layout which might be more power efficient.

The final board arrangement is shown in Figure 17.

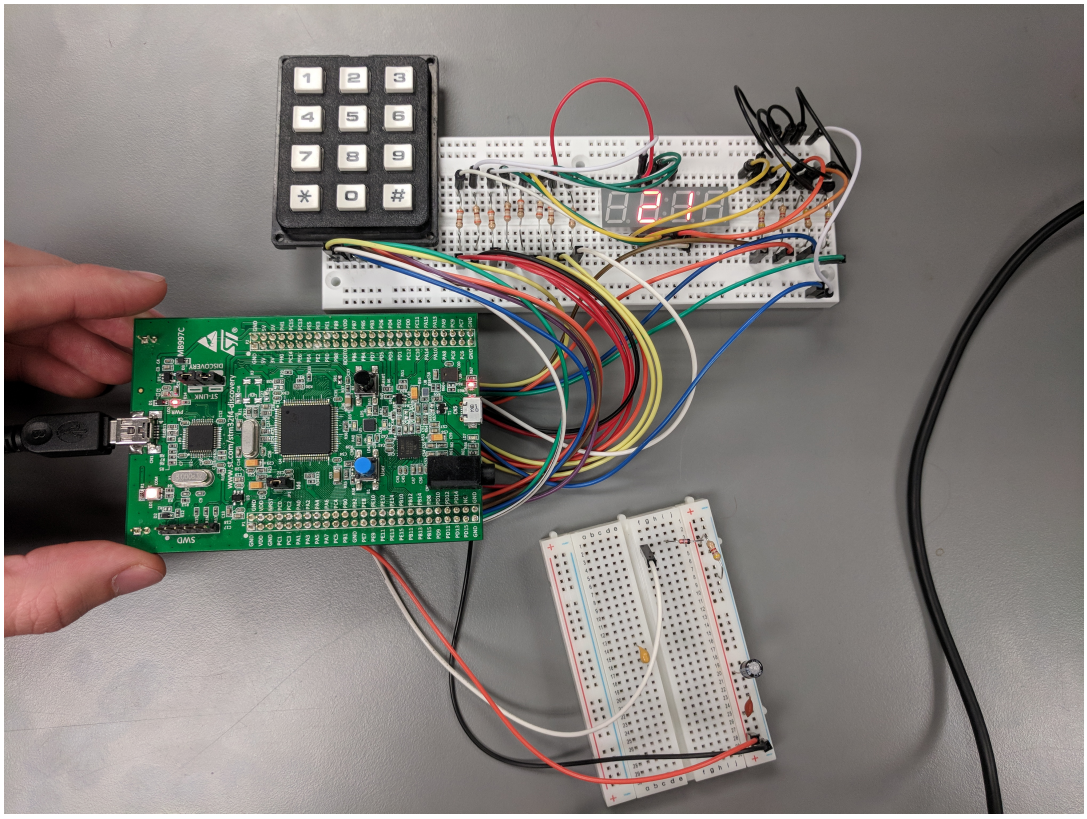


Figure 16: The final board setup.

5 conclusion

During the course of the last two labs, we created a system that produces a PWM signal in order to match a RMS voltage value in a circuit. The value is acquired through user input on a keypad. Based on a calculation of the time constant of our circuit, we were able to pick appropriate values for the PWM and ADC frequencies, which enabled our PWM controller to efficiently adjust the PWM duty cycle and match the target voltage. This system was designed to be as modular as possible, by isolating each logical section in a designated thread. Keypad, ADC and Display threads were created, which were responsible for managing user input, filtering and processing data, and driving the 7-segment display. The ADC thread implementation was later reverted back to an Interrupt-Service-Routine as a direct consequence of strange behaviour. Finally, this system is able to match a target voltage with 3 digits of precision in generally under 1.5s, which we deem to be acceptable in the context of this experiment. Despite limiting CPU usage in most cases, our system could be improved. Some additional improvements would include reducing system power consumption by removing the polling aspect of the keypad thread in favour of a combination of interrupts and a polling Thread.

6 appendix

List of Figures

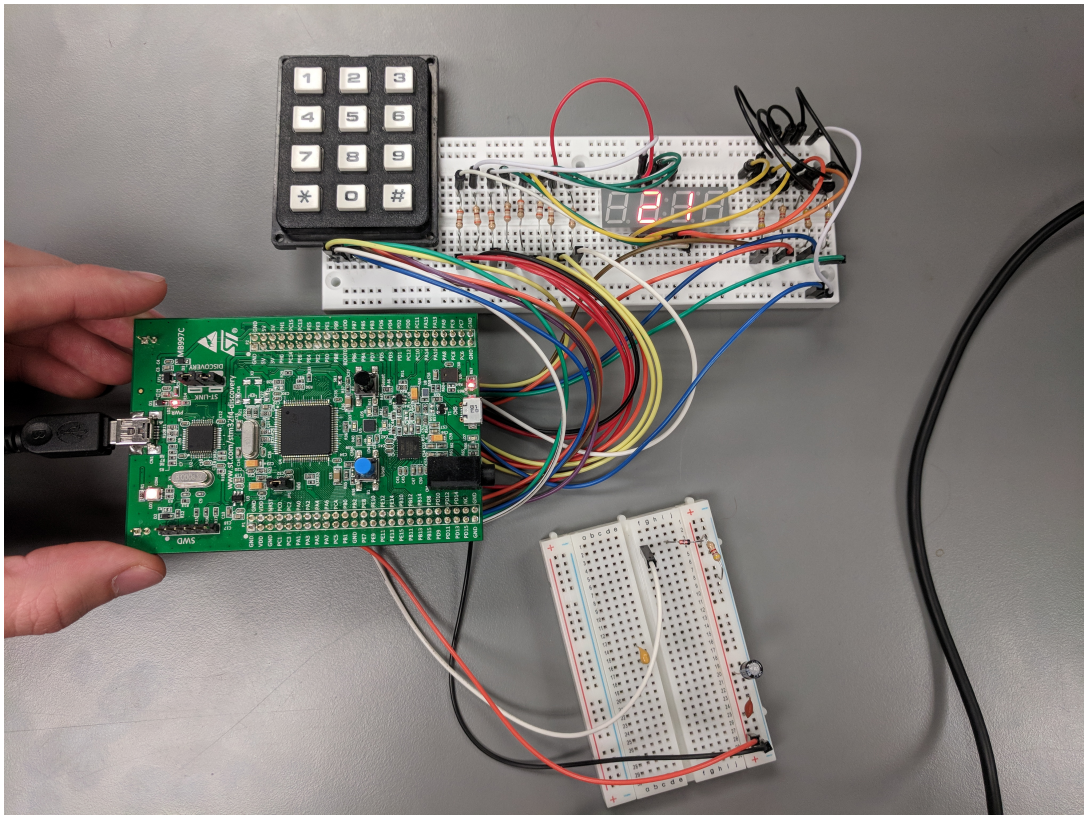


Figure 17: The final board setup.

Fabrice Normandin, 260636800
Marcel Morin, 260605670

Group 20
Due: February 19, 2018

ECSE 426 Lab 2 Report

1. Abstract

The goal of this lab was to successfully generate and acquire data using our STM32F407VG board's build in components, along with the implementation of data filtering and computation to produce information to be displayed to a user. With these challenges in mind we were tasked with implementing an efficient way to gather data sampled by our Analog to Digital Converter (ADC) without overloading our CPU all while filtering the data as it comes in. We came to realize that although an interrupt based approach could significantly reduce CPU usage, an even more efficient approach would be to use the DMA setting of our ADC, to have it place data at a predetermined location in memory. We later filtered and ran our required computations on that data. Knowledge of the Systick interrupt timer mechanics and of the ADC callback functions were crucial for the success of this lab and for an efficient solution with respect to CPU usage.

2. Problem Statement

Our gathered data required for this lab needed to firstly be generated in a controlled manner in order to be able to later test the effectiveness our filter and data calculations. This required us to generate an analog signal using the on board Digital to Analog Converter (DAC) and have our ADC sample this signal at 50Hz. Not only were we required to sample the signal 50 times a second, but we needed to do so in an interrupt based manner that required us to not constantly be polling our ADC for values.

With the collected ADC values our next challenge was to filter our data with an Finite Impulse Response filter to obtain a smoother data set and reduce noise. With this smoother data obtained we then needed to compute the minimum, maximum and Root Mean Squared values. The min and max values needed to be constantly updating over a window of the last 10 seconds of data acquired and the RMS needed to be the most updated value (we interpreted it as the RMS of the last second of data acquired). This was a challenge as all the while our ADC's values needed to be collected while we filtered, and computed on past values acquired simultaneously.

The final problem to implement was that our computed values needed to be translated onto a 7-segment LED display that would be able to switch between the minimum, maximum, and RMS value via the push of a button on our board. These values would also need to be updated and shown as a computed voltage rather than the digital value produced by the ADC.

3. Theory and Hypothesis

As previously mentioned, one of the major tasks of this lab was to successfully acquire analog data at a rate of 50Hz. We opted to use the ADC in a 12-bit resolution mode, although in theory, the values we were to display would only need 2 decimal digits of precision, hence we could have opted for a more efficient 8-bit resolution for the ADC. Since 3.0/(2^8) is about 0.01, a difference of one binary digit would register on the display properly, had we chosen 8 bits of resolution. However, with 12 bits, a change of one binary digit only represents 3.0/(2^12), or about 0.0007V, which does not affect the displayed value. Hence, the increase in resolution is not entirely justified. This decrease in precision would have enabled a higher sampling frequency, since the Successive-Approximation-Register (SAR) ADC would only need to produce 8 values instead of 12. In our case however, the ADC's sampling time was set to 460 clock cycles, making one full value reading about 5520 clock cycles long, which at our clock frequency of 186MHz, gives a maximum theoretical sampling frequency of about 33.7 KHz. Given our very low sampling frequency requirement of 50Hz, having such a slight increase in the sampling time was deemed inconsequential, and we proceeded onwards with other aspects of the Lab.

4. Implementation

The implementation of this labs was broken into several stages that we approached in order and build off of each other as we advanced throughout. These stages consisted of: DAC initialization, Blue button interrupt, ADC interrupt implementation, Systick interrupt ADC sampling frequency, ADC DMA configuration, data filtering with FIR, calculation of min/max/rms over sliding-window, and finally 7-segment LED display implementation.

Our process started with the implementation of the DAC which needed to generate a analog signal. Looking at Figure 2 in the appendix we can see that the DAC was configured to pin PA4 in CubeMX. This didn't interfere with other crucial components of our design and was closely located near our ADC input pin as to easily allow us to connect the two together. No further initialization was required for out DAC other than having it initialized by default with the generated CubeMX code in Keil. Starting the DAC was then a matter of Setting its value using HAL_DAC_SetValue and starting it using HAL_DAC_Start. SetValue was given the ADC handler as a parameter along with the channel that our DAC was outputting to, the data alignment, and the set 12 bit value to convert to analog.

In order to better understand how interrupts worked on our microprocessor we set out to implement the blue button interrupt handler so that when pressed an interrupt flag would be set letting our program know that the button was pressed without continuously polling its pin value to see if it had been set to high. We firstly needed to assign the GPIO_EXTI0 interrupt on the blue button pin PA0 as seen in Figure 2 in the appendix. With the pin assigned we then needed to enable its interrupt under the Nested Vectored Interrupt Controller (NVIC) settings in CubeMX. In Keil, the PA0 pin would have its associated EXTI0 handler interrupt priority set and enabled letting us use the called function EXTI0_IRQHandler() in the STM32f4xx_it.c file. At first we used this interrupt handler to change the four on-board LEDs to confirm our

implementation was correct, and only later into the lab did we change its function to change the displayed values of our computed data.

Our next stage involved initializing our ADC in interrupt mode. The settings initialized in CubeMX can be seen below in Figure 1. Our bits and data alignment needed to be specified. Scan continuous conversion mode was disabled as we were only working with one channel to convert our ADC data and did not need to convert multiple analog channels with the same ADC. Continuous conversion mode was disabled to enable us to control the sampling frequency of the ADC. We were also not using DMA at this stage in our experiment. Finally, *End of Conversion Selection* was set to "EOC flag at the end of all conversions", meaning that we could convert multiple values after one another without having to stop and restart our ADC. Before generating our ADC code for Keil we needed to make sure, like with the blue button interrupt, we enabled the NVIC setting for adc1 global interrupt enable. With these settings finalized in CubeMX we could implement how our ADC collected data in Keil.

ADC_Settings	
Clock Prescaler	PCLK2 divided by 4
Resolution	12 bits (15 ADC Clock cycles)
Data Alignment	Right alignment
Scan Conversion Mode	Disabled
Continuous Conversion Mode	Disabled
Discontinuous Conversion Mode	Disabled
DMA Continuous Requests	Disabled
End Of Conversion Selection	EOC flag at the end of all conversions

Figure 1 - ADC CubeMX settings

Gathering data from the interrupt routine of our ADC required the help of the System Clock timer (systick) interrupt and the implementation of a weakly defined function ("*HAL_ADC_ConvCpltCallback()*") in the HAL_ADC library. The systick interrupt was by default called every 6 microseconds. This was determined via experimentation in the Keil debug mode using timers to time each iteration of the systick interrupt and through the CubeMX clock configuration, Figure 3 in appendix, which shows the Cortex system timer to be divided by "/1". Changing the denominator to "/8" would cause the following clock config line in our generated code to change from HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000), divided by 1000, to divided by 8000. The HAL_SYSTICK_Config function sets the period of the systick interrupts per second and 168Mhz/1000 gives you 168000 period per interrupt, or 1ms per systick. The simple fix to get the ADC to sample properly would be to alter the denominator of the HAL_SYSTICK_Config function to get the systick interrupt to occur at a rate of 50Hz. This, however, isn't in a section of code that enables the user to write without having CubeMX regenerate over it were we to re-configure settings in CubeMX, and, were there to be other peripherals that later in the experiment were depended on the systick, the interrupt going at such a slow limiting rate would not be beneficial to our system. To solve this we kept the default systick configuration interrupt as initialized by CubeMX, and instead implemented a counter

within the systick interrupt to make sure that the ADC_Start_IT function was called at a rate of 50Hz, by counting up to 20 systick calls before firing the HAL_ADC_Start_IT() function and resetting the counter value to 0.

With the ADC now able to be called at the proper sampling rate, we could finally store our data into a buffer at every interrupt. To do this required us to define the function HAL_ADC_ConvCpltCallback in our main.c file. This function was called every time the ADC interrupt was triggered and ready to give out a converted value. The ADC_GetValue function would then be called within it and added to an ADCBuffer array that stored the gather data at every interrupt while incrementing its value after every ConvCpltCallback to fill up the array. We decided to store up to a second of data (50 samples) before looping back (mod 50) and overwriting the old entries at the start of the array. We were now able to continuously store data sampled from our ADC and to check whether the sampled data entered into our ADCBuffer was correct we connected the DAC pin PA4 to the ADC pin PA1 and could see through the debug window that our retrieved values were the same +- 5 of what our 12 bit DAC_SetValue was.

Before continuing with the data calculation, we decided to implement DMA instead of the ADC interrupt as a means to sample our values. To do this required us to go back into CubeMX and enable DMA requests from our ADC peripheral. Doing this would generate a DMA global interrupt in Keil that we realized would trigger when the DMA buffer was halfway and completely full when the HalfConvCpltCallback and ConvCpltCallback functions would be called respectively. In doing this we did not need to have the ADC_GetValue inside the ConvCpltCallback function anymore as DMA was taking care of that. Instead once the ConvCpltCallback was called we just for looped through the full ADCBuffer and applied our FIR filter function per loop iteration and respectively added the filtered value to the filtered_ADCBuffer array. Making sure DMA was in circular mode would mean that DMA would loop back around to the start of the memory we allocated it. Replacing HAL_ADC_Start_IT in our main right before the while loop with HAL_ADC_Start_DMA and passing through the pointer to our ADCBuffer and the buffer size is all that is needed to start the DMA process. The systick interrupt handler still controls the rate at which we are sampling our data and filling up the buffer so no changes were required there. However, an issue arose after the first second of data had been acquired, as the DMA seemed to have stopped. Although we considered looking into using the *DMA_Continuous_Request* setting, we, in the interest of time, instead used ADC_Stop_DMA() followed by ADC_Start_DMA() once every second, which would successfully gather data for the next second with DMA.

The next stage of our experiment was to filter the incoming data using our previous lab's FIR filter. This filter takes the received ADC sampled value and applies a filter over the previous 4 gathered samples by adding all 5 values with a weight of 0.2 on each value. We implemented this using a buffer in a circular queue and added the newest sampled element at the head pointer (making sure to mod5 back around to overwrite the 6th oldest value with the newest sampled value). We then added up all the elements going from the head to the tail and returned

that as the filtered value of our FIR function. This value was then added in our ADCBuffer instead of the original value the ADC_GetValue function returned.

Then later came the task of storing the Max and Min values of the past 10 seconds, along with the latest RMS value, which we assumed to be that of the last second. We implemented this by first computing the min, max and RMS of the last second's data using the asm_math assembly code from Lab 1, then stored the results in an array, using head and tail pointers in order to create a circular queue. This array, with a capacity of 10 for max and min values (one array per value) had a new set of Max and Min inserted every second. The RMS value was also stored and overwritten every second. During each call of our *adc_buffer_full_callback()* function, the min and max of the last 10 seconds were also computed, and a global static integer variable (*display_mode*) was used in order to set *displayed_value*, a global static float variable, to the required value (either min, max or RMS), depending on the current display mode, which was changed using the blue button, and indicated with a set of three LEDs, which would alternate depending on which mode was currently active.

Then, once the *display_mode* and *displayed_value* were set, a simple *refresh_display()* method was implemented, which gets called by the SysTick interrupt handler at a customizable frequency (generally about 200Hz). This method was responsible for converting the float *displayed_value* to a set of three digits, and then alternate in showing one of these three digits on the 7-segment display. To achieve this efficiently without using delays, our simple approach consisted of calculating the segments required for each digit, and then turning on one of the digits once out of three calls to the function, alternating from digits 0, 1 and 2. A set of very helpful macros were implemented within the "*heads_up_display.h"* file, which greatly facilitated setting the GPIO pins. The Output Data Register (ODR) was used to set multiple GPIO pins at once, and also helped in having the program be efficient by bypassing the hardware abstraction layer.

5. Testing & Observations

In order to test the conversion accuracy of our ADC, we used an oscilloscope to view the voltage that our DAC was outputting, as well as that of the 3V VDD pins on the board. At first, we were pleased to see the values approximately line up, with the ADC value chosen to be about half the range (2048 out of 4095), we were seeing about 1.6 volts, when the oscilloscope was showing around 1.53V. We were then thankfully informed that our assumption of 3.3V for the maximum DAC voltage was wrong, as the VDD lines were at 3V. By simply fixing the coefficient in our *DigitalToAnalogValue* function, we then got the value we displayed to correspond precisely to that shown by the oscilloscope.

6. Conclusion

Through the efficient use of interrupts and DMA we were able to sample our DAC's set value via our ADC and then compute the minimum, maximum, and rms values of the past 10

seconds. We made sure to have each stage properly working, as described in our implementation section, before moving on to the next step, which made it easy to debug our code to find the source of errors. This experiment shed light on the importance of interrupts and lightening the CPU load for repetitive tasks that could be done by peripheral controllers. This paves the way for future labs giving the proper understanding of the ADC, DAC, GPIO pins and interrupt handlers that will be useful to implement in future experiments.

APPENDIX

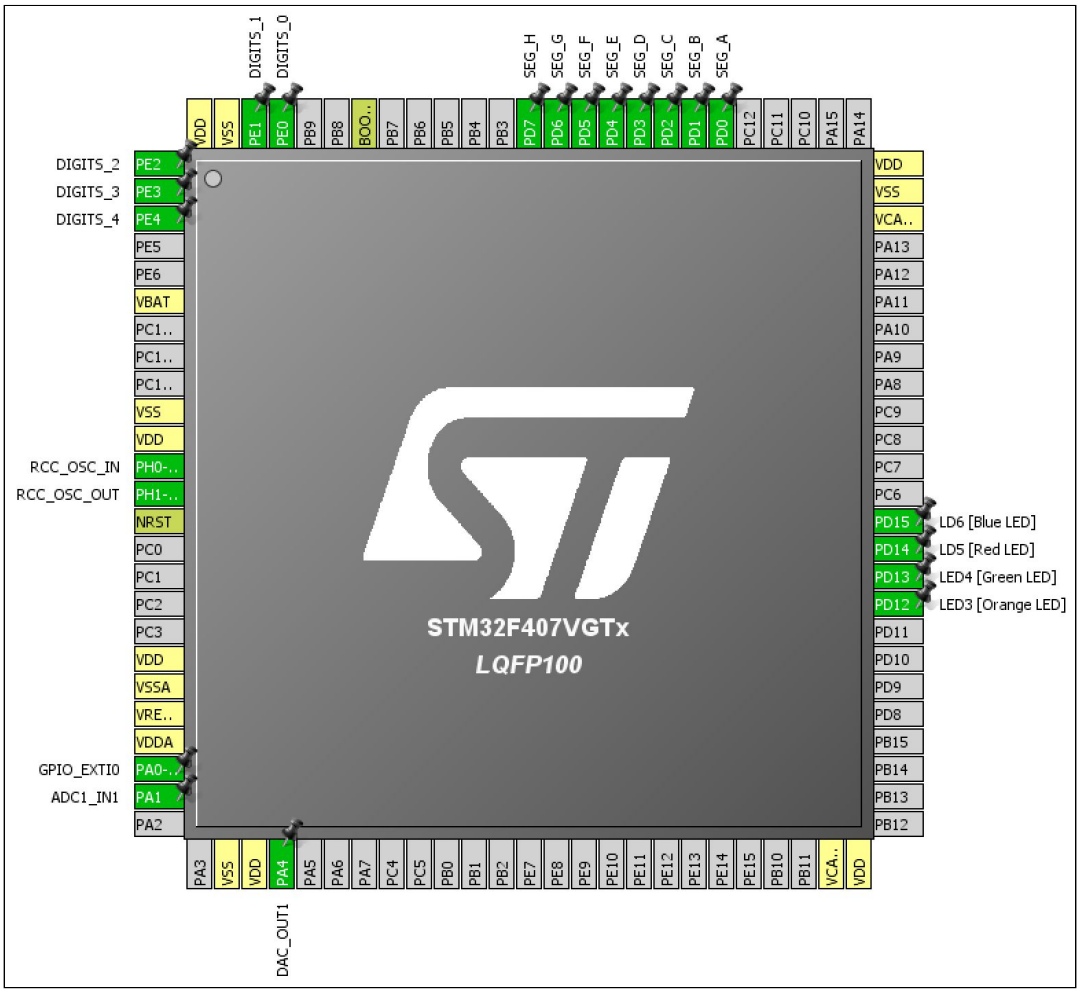


Figure 2 - CubeMX final pin layout

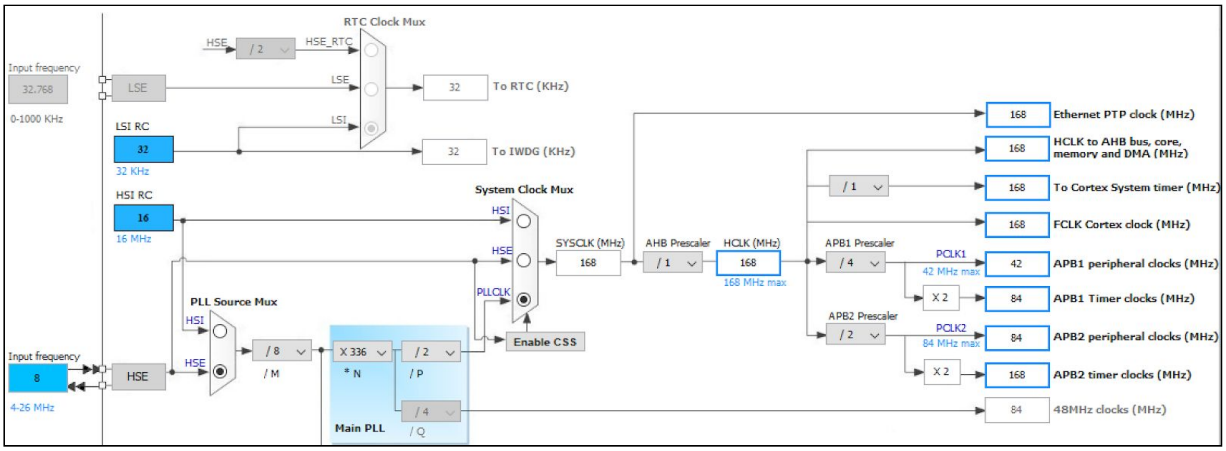


Figure 3 - CubeMX clock configuration settings

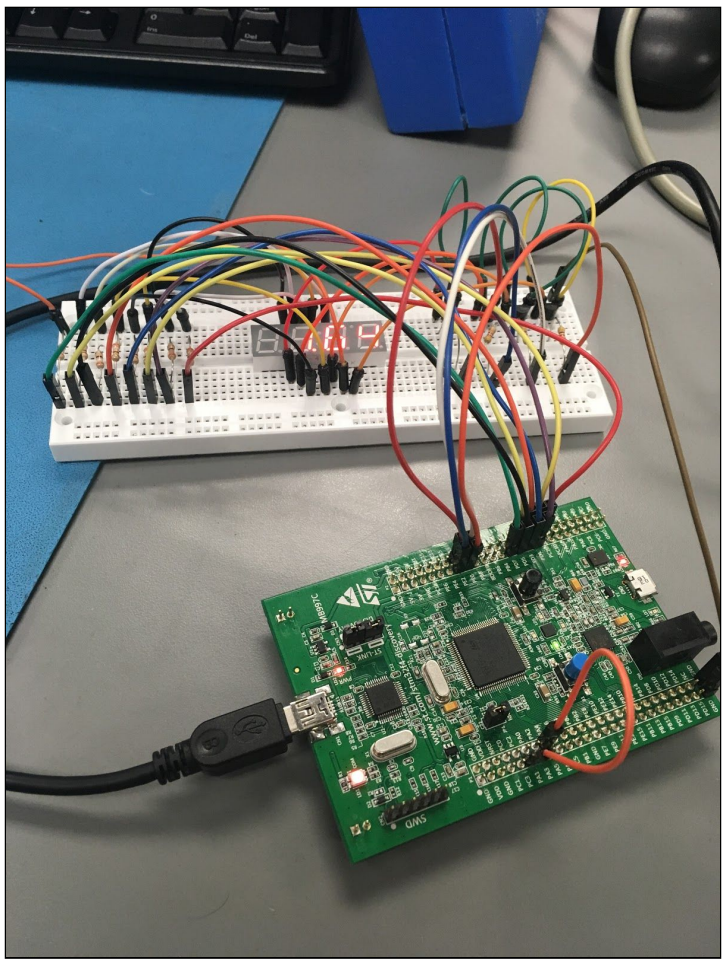


Figure 4 - Final setup of the experiment with working 7-segment display

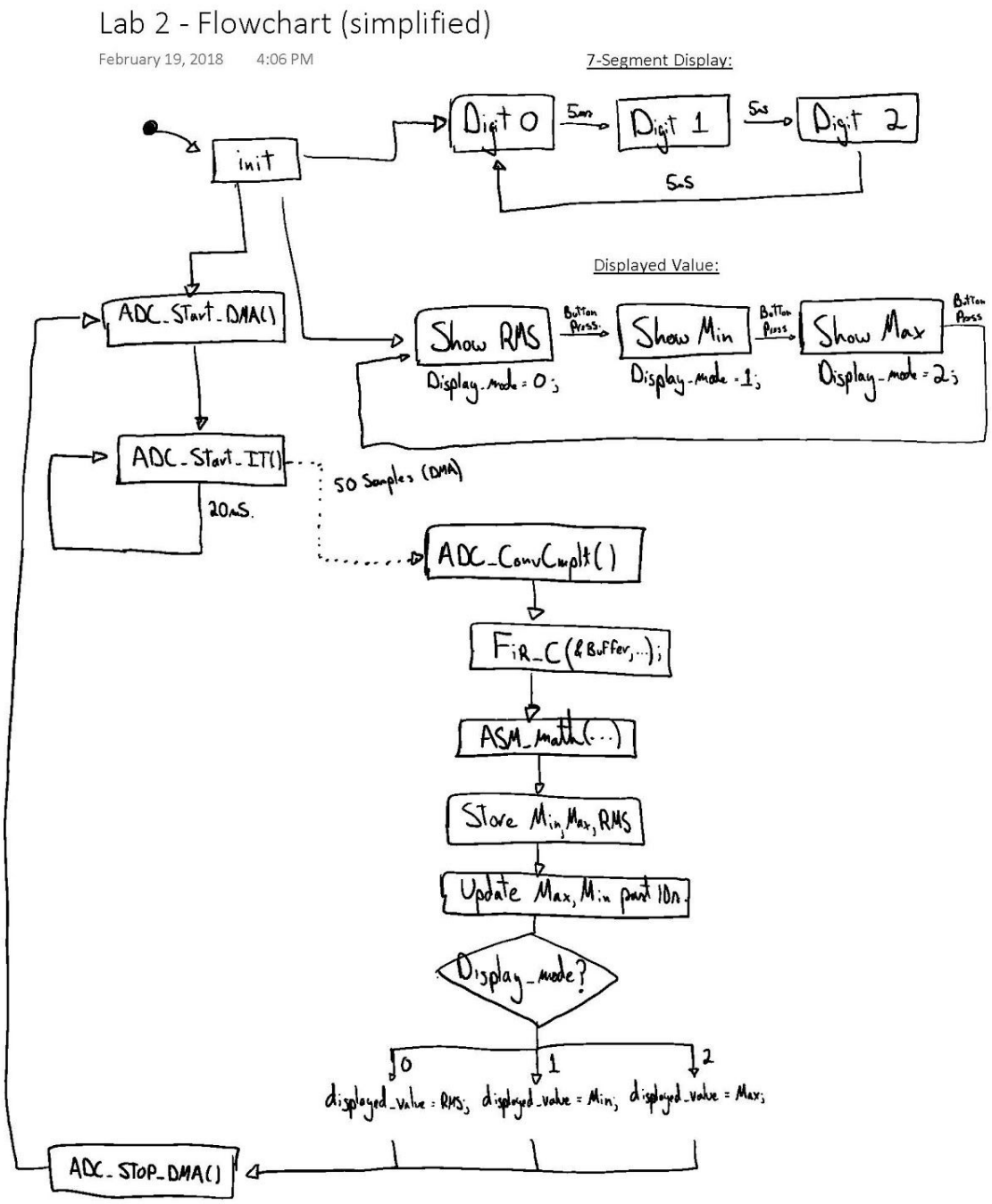


Figure 5 - Simplified flowchart